MODULE *DistributedLock*

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

The set of clients
CONSTANT *Clients*

Client states
CONSTANTS *Active*, *Inactive*

Message types
CONSTANT *LockRequest*, *LockResponse*, *TryLockRequest*, *TryLockResponse*, *UnlockRequest*, *UnlockResponse*

An empty constant
CONSTANT *Nil*

The current lock holder
VARIABLE *lock*

The lock queue
VARIABLE *queue*

The current lock *ID*
VARIABLE *id*

Session states
VARIABLE *sessions*

$serverVars \triangleq \langle lock, id, queue, sessions \rangle$

Client states
VARIABLE *clients*

$clientVars \triangleq \langle clients \rangle$

Client requests
VARIABLE *requests*

Server responses
VARIABLE *responses*

Variable to track the total number of messages sent for use in state constraints when model checking
VARIABLE *messageCount*

$messageVars \triangleq \langle requests, responses, messageCount \rangle$

The invariant checks that:
* No client can hold more than one lock at a time
* No two clients hold a lock with the same *ID*

* The lock is held by an active session

Note that more than one client may believe itself to hold the lock at the same time, *e.g.* if a client's session has expired but the client hasn't been notified, but lock *IDs* must be unique and monotonically increasing.

$$
\begin{aligned}
&\mathit{TypeInvariant} \triangleq \\
&\quad \land \forall\, c \in \text{DOMAIN } \mathit{clients} : \mathit{Cardinality}(\mathit{clients}[c].\mathit{locks}) \in 0 \,..\, 1 \\
&\quad \land \forall\, c1,\ c2 \in \text{DOMAIN } \mathit{clients} : c1 \neq c2 \Rightarrow \mathit{Cardinality}(\mathit{clients}[c1].\mathit{locks} \cap \mathit{clients}[c2].\mathit{locks}) = 0 \\
&\quad \land \mathit{lock} \neq \mathit{Nil} \Rightarrow \mathit{sessions}[\mathit{lock}.\mathit{client}].\mathit{state} = \mathit{Active}
\end{aligned}
$$

Returns a sequence with the head removed

$$\mathit{Pop}(q) \triangleq \mathit{SubSeq}(q,\ 2,\ \mathit{Len}(q))$$

Sends a message on the given client's channel

$$
\begin{aligned}
&\mathit{Send}(m,\ c) \triangleq \\
&\quad \land \mathit{requests}' = [\mathit{requests} \text{ EXCEPT } ![c] = \mathit{Append}(\mathit{requests}[c],\ m)] \\
&\quad \land \mathit{messageCount}' = \mathit{messageCount} + 1 \\
&\quad \land \text{UNCHANGED } \langle \mathit{responses} \rangle
\end{aligned}
$$

Removes a message from the given client's channel

$$
\begin{aligned}
&\mathit{AcceptRequest}(m,\ c) \triangleq \\
&\quad \land \mathit{requests}' = [\mathit{requests} \text{ EXCEPT } ![c] = \mathit{Pop}(\mathit{requests}[c])] \\
&\quad \land \mathit{messageCount}' = \mathit{messageCount} + 1 \\
&\quad \land \text{UNCHANGED } \langle \mathit{responses} \rangle
\end{aligned}
$$

Removes a message from the given server's channel

$$
\begin{aligned}
&\mathit{AcceptResponse}(m,\ c) \triangleq \\
&\quad \land \mathit{responses}' = [\mathit{responses} \text{ EXCEPT } ![c] = \mathit{Pop}(\mathit{responses}[c])] \\
&\quad \land \mathit{messageCount}' = \mathit{messageCount} + 1 \\
&\quad \land \text{UNCHANGED } \langle \mathit{requests} \rangle
\end{aligned}
$$

Removes the last message from the client's channel and appends a message to the given server's channel

$$
\begin{aligned}
&\mathit{Reply}(m,\ c) \triangleq \\
&\quad \land \mathit{requests}' = [\mathit{requests} \text{ EXCEPT } ![c] = \mathit{Pop}(\mathit{requests}[c])] \\
&\quad \land \mathit{responses}' = [\mathit{responses} \text{ EXCEPT } ![c] = \mathit{Append}(\mathit{responses}[c],\ m)] \\
&\quad \land \mathit{messageCount}' = \mathit{messageCount} + 1
\end{aligned}
$$

Handles a lock request. If the lock is not currently held by another process, the lock is granted to the client. If the lock is held by a process, the request is added to a queue.

$$
\begin{aligned}
&\mathit{HandleLockRequest}(m,\ c) \triangleq \\
&\quad \lor \land \mathit{sessions}[c].\mathit{state} \neq \mathit{Active} \\
&\qquad \land \mathit{AcceptRequest}(m,\ c) \\
&\qquad \land \text{UNCHANGED } \langle \mathit{clientVars},\ \mathit{serverVars} \rangle \\
&\quad \lor \land \mathit{sessions}[c].\mathit{state} = \mathit{Active} \\
&\qquad \land \mathit{lock} = \mathit{Nil}
\end{aligned}
$$

```
        ∧ lock′ = m @@ ("client" :> c)
        ∧ id′ = id + 1
        ∧ Reply([type ↦ LockResponse, acquired ↦ TRUE, id ↦ id′], c)
        ∧ UNCHANGED ⟨queue, sessions, clientVars⟩
    ∨ ∧ sessions[c].state = Active
      ∧ lock ≠ Nil
      ∧ queue′ = Append(queue, m @@ ("client" :> c))
      ∧ AcceptRequest(m, c)
      ∧ UNCHANGED ⟨lock, id, sessions, clientVars⟩
```

Handles a *tryLock* request. If the lock is not currently held by another process, the lock is granted to the client. Otherwise, the request is rejected.

```
HandleTryLockRequest(m, c) ≜
    ∨ ∧ sessions[c].state ≠ Active
      ∧ AcceptRequest(m, c)
      ∧ UNCHANGED ⟨clientVars, serverVars⟩
    ∨ ∧ sessions[c].state = Active
      ∧ lock = Nil
      ∧ lock′ = m @@ ("client" :> c)
      ∧ id′ = id + 1
      ∧ Reply([type ↦ LockResponse, acquired ↦ TRUE, id ↦ id′], c)
      ∧ UNCHANGED ⟨queue, sessions, clientVars⟩
    ∨ ∧ sessions[c].state = Active
      ∧ lock ≠ Nil
      ∧ Reply([type ↦ LockResponse, acquired ↦ FALSE], c)
      ∧ UNCHANGED ⟨clientVars, serverVars⟩
```

Handles an unlock request. If the lock is currently held by the given client, it will be unlocked. If any client's requests are pending in the queue, the next lock request will be removed from the queue and the lock will be granted to the requesting client.

```
HandleUnlockRequest(m, c) ≜
    ∨ ∧ sessions[c].state ≠ Active
      ∧ AcceptRequest(m, c)
      ∧ UNCHANGED ⟨clientVars, serverVars⟩
    ∨ ∧ sessions[c].state = Active
      ∧ lock = Nil
      ∧ AcceptRequest(m, c)
      ∧ UNCHANGED ⟨clientVars, serverVars⟩
    ∨ ∧ sessions[c].state = Active
      ∧ lock ≠ Nil
      ∧ lock.client = c
      ∧ lock.id = m.id
      ∧ ∨ ∧ Len(queue) > 0
          ∧ LET next ≜ Head(queue)
            IN
                ∧ lock′ = next
```

```
                    ∧ id′ = id + 1
                    ∧ queue′ = Pop(queue)
                    ∧ Reply([type ↦ LockResponse, acquired ↦ TRUE, id ↦ id′], next.client)
                    ∧ UNCHANGED ⟨sessions⟩
          ∨ ∧ Len(queue) = 0
            ∧ lock′ = Nil
            ∧ AcceptRequest(m, c)
            ∧ UNCHANGED ⟨queue, id, sessions⟩
    ∧ UNCHANGED ⟨clientVars⟩
```

Expires a client's session. If the client currently holds the lock, the lock will be released and the lock will be granted to another client if possible. Additionally, pending lock requests from the client will be removed from the queue.

```
ExpireSession(c) ≜
    ∧ sessions[c].state = Active
    ∧ sessions′ = [sessions EXCEPT ![c].state = Inactive]
    ∧ LET isActive(m) ≜ sessions′[m.client].state = Active
      IN
          IF lock ≠ Nil ∧ lock.client = c THEN
              LET q ≜ SelectSeq(queue, isActive)
              IN
                  ∨ ∧ Len(q) > 0
                    ∧ lock′ = Head(q)
                    ∧ id′ = id + 1
                    ∧ queue′ = Pop(q)
                    ∧ Send([type ↦ LockResponse, acquired ↦ TRUE, id ↦ id′], lock′.client)
                  ∨ ∧ Len(queue) = 0
                    ∧ lock′ = Nil
                    ∧ queue′ = ⟨⟩
                    ∧ UNCHANGED ⟨id, messageVars⟩
           ELSE
              ∧ queue′ = SelectSeq(queue, isActive)
              ∧ UNCHANGED ⟨lock, id, messageVars⟩
    ∧ UNCHANGED ⟨clientVars⟩
```

Closes a client's expired session. This is performed in a separate step to model the time between the cluster expiring a session and the client being notified. A client can close its session either before or after it's expired by the cluster. Once the client believes its session has expired, its locks are removed, meaning a client can also believe itself to hold a lock after its session has expired in the cluster.

```
CloseSession(c) ≜
    ∧ clients[c].state = Active
    ∧ clients′ = [clients EXCEPT ![c].state = Inactive,
                                 ![c].locks = {}]
    ∧ UNCHANGED ⟨serverVars, messageVars⟩
```

Sends a lock request to the cluster with a unique *ID* for the client.

$$
\begin{array}{l}
Lock(c) \triangleq \\
\quad \land\ clients[c].state = Active \\
\quad \land\ Send([type \mapsto LockRequest,\ id \mapsto clients[c].next],\ c) \\
\quad \land\ clients' = [clients\ \text{EXCEPT}\ ![c].next = clients[c].next + 1] \\
\quad \land\ \text{UNCHANGED}\ \langle serverVars \rangle
\end{array}
$$

Sends a try lock request to the cluster with a unique *ID* for the client.

$$
\begin{array}{l}
TryLock(c) \triangleq \\
\quad \land\ \ clients[c].state = Active \\
\quad \land\ \ Send([type \mapsto TryLockRequest,\ id \mapsto clients[c].next],\ c) \\
\quad \land\ \ clients' = [clients\ \text{EXCEPT}\ ![c].next = clients[c].next + 1] \\
\quad \land\ \ \text{UNCHANGED}\ \langle serverVars \rangle
\end{array}
$$

Sends an unlock request to the cluster if the client is active and current holds a lock.

$$
\begin{array}{l}
Unlock(c) \triangleq \\
\quad \land\ clients[c].state = Active \\
\quad \land\ Cardinality(clients[c].locks) > 0 \\
\quad \land\ Send([type \mapsto UnlockRequest,\ id \mapsto \text{CHOOSE}\ l \in clients[c].locks : \text{TRUE}],\ c) \\
\quad \land\ clients' = [clients\ \text{EXCEPT}\ ![c].locks = clients[c].locks \setminus \{\text{CHOOSE}\ l \in clients[c].locks : \text{TRUE}\}] \\
\quad \land\ \text{UNCHANGED}\ \langle serverVars \rangle
\end{array}
$$

Handles a lock response from the cluster. If the client's session is expired, the response is ignored. If the lock was acquired successfully, it's added to the client's lock set.

$$
\begin{array}{l}
HandleLockResponse(m,\ c) \triangleq \\
\quad \land\ \lor\ \land\ clients[c].state = Inactive \\
\qquad\quad \land\ \text{UNCHANGED}\ \langle clientVars,\ serverVars \rangle \\
\qquad \lor\ \land\ clients[c].state = Active \\
\qquad\quad \land\ m.acquired \\
\qquad\quad \land\ clients' = [clients\ \text{EXCEPT}\ ![c].locks = clients[c].locks \cup \{m.id\}] \\
\qquad\quad \land\ \text{UNCHANGED}\ \langle serverVars \rangle \\
\qquad \lor\ \land\ clients[c].state = Active \\
\qquad\quad \land\ \neg m.acquired \\
\qquad\quad \land\ \text{UNCHANGED}\ \langle clientVars,\ serverVars \rangle \\
\quad \land\ AcceptResponse(m,\ c)
\end{array}
$$

Receives a message from/to the given client from the head of the client's message queue.

$$
\begin{array}{l}
Receive(c) \triangleq \\
\quad \lor\ \ \land\ Len(requests[c]) > 0 \\
\qquad \land\ \text{LET}\ message \triangleq Head(requests[c]) \\
\qquad\quad \text{IN} \\
\qquad\qquad \lor\ \land\ message.type = LockRequest
\end{array}
$$

$$
\begin{array}{l}
\quad\quad\quad\quad\quad\quad\quad\quad \land\ HandleLockRequest(message,\ c) \\
\quad\quad\quad\quad\quad\quad\quad \lor\ \land\ message.type = TryLockRequest \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \land\ HandleTryLockRequest(message,\ c) \\
\quad\quad\quad\quad\quad\quad\quad \lor\ \land\ message.type = UnlockRequest \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \land\ HandleUnlockRequest(message,\ c) \\
\quad\quad \lor\ \land\ Len(responses[c]) > 0 \\
\quad\quad\quad\ \land \text{LET}\ message \triangleq Head(responses[c]) \\
\quad\quad\quad\quad \text{IN} \\
\quad\quad\quad\quad\quad\quad\quad \lor\ \land\ message.type = LockResponse \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \land\ HandleLockResponse(message,\ c)
\end{array}
$$

Initial state predicate

$$
\begin{array}{l}
Init \triangleq \\
\quad \land\ requests = [c \in Clients \mapsto \langle\rangle] \\
\quad \land\ responses = [c \in Clients \mapsto \langle\rangle] \\
\quad \land\ messageCount = 0 \\
\quad \land\ lock = Nil \\
\quad \land\ queue = \langle\rangle \\
\quad \land\ id = 0 \\
\quad \land\ clients = [c \in Clients \mapsto [state \mapsto Active,\ locks \mapsto \{\},\ next \mapsto 1]] \\
\quad \land\ sessions = [c \in Clients \mapsto [state \mapsto Active]]
\end{array}
$$

Next state predicate

$$
\begin{array}{l}
Next \triangleq \\
\quad \lor \exists\, c \in \text{DOMAIN}\ clients : Receive(c) \\
\quad \lor \exists\, c \in \text{DOMAIN}\ clients : Lock(c) \\
\quad \lor \exists\, c \in \text{DOMAIN}\ clients : TryLock(c) \\
\quad \lor \exists\, c \in \text{DOMAIN}\ clients : Unlock(c) \\
\quad \lor \exists\, c \in \text{DOMAIN}\ clients : ExpireSession(c) \\
\quad \lor \exists\, c \in \text{DOMAIN}\ clients : CloseSession(c)
\end{array}
$$

The specification includes the initial state predicate and the next state

$$Spec \triangleq Init \land \Box[Next]_{\langle serverVars,\ clientVars,\ messageVars \rangle}$$

---

* Modification History
* Last modified Sun *Jan* 28 10:20:37 *PST* 2018 by *jordanhalterman*
* Created *Fri Jan* 26 13:12:01 *PST* 2018 by *jordanhalterman*